AD-A251 604

ITATION PAGE

Form Approved
OPM No.

(2)

1.	ORT	3. REPORT TYPE AND DATES
		Final: 06 Mar 1992

4. TITLE AND
Validation Summary Report: NATO SWG on APSE Compiler for VAX/VMS to MC68020, Version VCM1.82-02, VAX 8350 under VMS 5.4-1 with CAIS 5.5E (Host) Motorola MVME 133XT(Target), 920306I1.11248

5. FUNDING

6.
IABG-AVF
Ottobrunn, Federal Republic of Germany

7. PERFORMING ORGANIZATION NAME(S) AND
IABG-AVF, Industrieanlagen-Betriebsgeselschaft
Dept. SZT/ Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION
IABG-VSR 102

9. SPONSORING/MONITORING AGENCY NAME(S) AND
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY

11. SUPPLEMENTARY

12a. DISTRIBUTION/AVAILABILITY
Approved for public release; distribution unlimited.

12b. DISTRIBUTION

13. *(Maximum 200*
NATO SWG on APSE Compiler for VAX/VMS to MC68020, Version VCM1.82-02, VAX 8350 under VMS 5.4-1 with CAIS 5.5E (Host) Motorola MVME 133XT(MC68020 bare machine)(Target), ACVC 1.11.

14. SUBJECT
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A,

15. NUMBER OF

16. PRICE

17. SECURITY CLASSIFICATION	18. SECURITY	19. SECURITY CLASSIFICATION	20. LIMITATION OF
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

NSN

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std.

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on March 06, 1992.

Compiler Name and Version:
NATO SWG on APSE Compiler for VAX/VMS to MC68020
Version 1.82-02

Host Computer System:
VAX 8350 under VMS Version 5.4-1
with CAIS Version 5.5E

Target Computer System:
Motorola MVME 133XT (MC68020 bare machine)

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 920306I1.11248 is awarded to Alsys. This certificate expires on 1 June, 1993.

This report has been reviewed and is approved.

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

AVF Control Number: IABG-VSR 102
06 March 1992

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 920306I1.11248
NATO SWG on APSE Compiler
for VAX/VMS to MC68020
Version VCM1.82-02
VAX 8350 under VMS 5.4-1 with CAIS 5.5E Host
Motorola MVME 133XT (MC68020 bare machine) Target

== based on TEMPLATE Version 91-05-08 ==

Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
Germany

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer: Alsys GmbH & Co. KG

Certificate Awardee: Alsys / German MoD

Ada Validation Facility: IABG mbH, Germany

ACVC Version: 1.11

Ada Implementation:

NATO SWG on APSE Compiler for VAX/VMS to MC68020 Version VCM1.82-02

Host Computer System:
VAX 8350 under VMS Version 5.4-1, with CAIS Version 5.5E

Target Computer System: Motorola MVME 133XT (MC68020) (bare machine)

Declaration:

We, the undersigned, declare that we have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

[signature] 11.3.92

Customer Signature **Date**

Alsys GmbH & Co. KG
7500 Karlsruhe 51 Am Rüppurrer Schloß 7
Tel. 07 21/66 30 25 Fax 07 21/66 75 64

[signature] 11.3.92

Certificate Awardee Signature **Date**

BUNDESMINISTER DER VERTEIDIGUNG

16. 03. 92

Im Auftrag Wilde Fregattenkapitän

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for

this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1), and possibly removing some inapplicable tests (see section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of

	several inter-connected units.
Conformity	Fulfillment by a product, process, or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 02 August 1991.

E28005C	B28006C	C32203A	C34006D	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	B41308B	C43004A
C45114A	C45346A	C45612A	C45612B	C45612C	C45651A
C46022A	B49008A	B49008B	A74006A	C74308A	B83022B
B83022H	B83025B	B83025D	B83026B	C83026A	C83041A
B85001L	C86001F	C94021A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
AD7206A	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Approved Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Approved Ada Commentaries are included as appropriate.

B22005A..C and B22005P (4 tests), respectively, check that the control characters SOH, STX, ETX, and DLE are illegal when outside of character literals, string literals, and comments; for this implementation those characters have a special meaning to the underlying system such that the test file is altered before being passed to the compiler. (See section 2.3.)

The following 159 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113O..Y (11 tests) (*)	C35705O..Y (11 tests)
C35706O..Y (11 tests)	C35707O..Y (11 tests)
C35708O..Y (11 tests)	C35802O..Z (12 tests)
C45241O..Y (11 tests)	C45321O..Y (11 tests)
C45421O..Y (11 tests)	C45521O..Z (12 tests)
C45524O..Z (12 tests)	C45621O..Z (12 tests)
C45641O..Y (11 tests)	C46012O..Z (12 tests)

(*) C24113W..Y (3 tests) contain lines of length greater than 255 characters which are not supported by this implementation.

The following 20 tests check for the predefined type LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C41401A checks that CONSTRAINT_ERROR is raised upon the evaluation of various attribute prefixes; this implementation derives the attribute values from the subtype of the prefix at compilation time, and thus does not evaluate the prefix or raise the exception. (See Section 2.3.)

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater. For this implementation, MAX_MANTISSA is less than 47.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is TRUE.

B86001Y checks for a predefined fixed-point type other than DURATION; for this implementation, there is no such type.

C96005B checks for values of type DURATION'BASE that are outside the range of DURATION. There are no such values for this implementation.

CD1009C uses a representation clause specifying a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types; this implementation does not support such sizes.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE_CODE,

The following 264 tests check operations on sequential, text, and direct access files; this implementation does not support external files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)
CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

CE2103A, CE2103B, and CE3107A expect that NAME_ERROR is raised when an attempt is made to create a file with an illegal name; this implementation does not support the creation of external files and so raises USE_ERROR. (See section 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 30 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24009A	B29001A	B38003A	B38009A	B38009B
B91001H	BC2001D	BC2001E	BC3204B	BC3205B	BC3205D

B22005A..C and B22005P (4 tests) were graded inapplicable by Evaluation Modification as directed by the AVO. These tests, respectively, check that control characters SOH, STX, ETX, and DLE are illegal outside of character literals, string literals, and comments. This implementation's underlying CAIS system gives special meaning to each of these control characters such that their effect is to alter the test files in a way that defeats the test objectives--either the characters alone, or together with any text that follows them on the line, are not passed to the compiler. Hence, B22005B and B22005P compile without error, while the other tests have syntactic errors introduced by the loss of test text.

B25002A, B26005A, and B27005A were graded passed by Evaluation Modification as directed by the AVO. These tests check that control characters SOH, STX, ETX, and DLE are illegal within of character literals, string literals, and comments, respectively. This implementation's underlying CAIS system gives special meaning to each of these control characters such that their effect is to alter the test files in the following way: these characters, and except in the case of DLE any text that follows them on the line, are not passed to the compiler. The tests were thus graded without regard for the lines that contained one of these four control characters.

C34007P and C34007S were graded passed by Evaluation Modification as directed by the AVO. These tests include a check that the evaluation of the selector "all" raises CONSTRAINT_ERROR when the value of the object is null. This implementation determines the result of the equality tests at lines 207 and 223, respectively, based on the subtype of the object; thus, the selector is not evaluated and no exception is raised, as allowed by LRM 11.6(7). The tests were graded passed given that their only output from REPORT.FAILED was the message "NO EXCEPTION FOR NULL.ALL - 2".

C41401A was graded inapplicable by Evaluation Modification as directed by the AVO. This test checks that the evaluation of attribute prefixes that denote variables of an access type raises CONSTRAINT_ERROR when the value of the variable is null and the attribute is appropriate for an array or task type. This implementation derives the array attribute values from the subtype; thus, the prefix is not evaluated and no exception is raised, as allowed by LRM 11.6(7), for the checks at lines 77, 87, 97, 108, 121, 131, 141, 152, 165, & 175.

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE

(REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package Report's body, and thus the packages' calls to function REPORT.IDENT_INT at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

BC3204C..D and BC3205C..D (4 tests) were graded passed by Evaluation Modification as directed by the AVO. These tests are expected to produce compilation errors, but this implementation compiles the units without error; all errors are detected at link time. This behavior is allowed by AI-00256, as the units are illegal only with respect to units that they do not depend on.

CE2103A, CE2103B, and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when USE_ERROR is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical and sales information about this Ada implementation system, see:

Alsys GmbH & Co. KG
Am Rüppurrer Schloß 7
W-7500 Karlsruhe 51
Germany
Tel. +49 721 883025

Testing of this Ada implementation was conducted at the AVF's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum

precision (item e; see section 2.2), and those that depend on the support of a file system -- if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3593	
b) Total Number of Withdrawn Tests	95	
c) Processed Inapplicable Tests	59	
d) Non-Processed I/O Tests	264	
e) Non-Processed Floating-Point Precision Tests	159	
f) Total Number of Inapplicable Tests	482	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

ACVC 1.11 was run at IABG's premises as follows: With the customer's macro parameter file the customised ACVC 1.11 was produced. Then CAIS version 5.5E as supplied by the customer was loaded and installed on the candidate VAX 8350 computer. Next the basic CAIS node model and the candidate Ada implementation were installed. Then the full set of tests was processed using command scripts provided by the customer and reviewed by the validation team. Tests were processed in two steps. In the first step tests were compiled and linked as appropriate, producing listings and possibly load modules. In the second step load modules were downloaded to the target computer via a serial communications line using a command script provided by the customer and reviewed by the validation team and executed on the target. The results were captured on the host system. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options.

Compilation was made using the following parameter settings:

```
SOURCE  => "'CURRENT_USER'DOT(SRC)"
LIBRARY => "'CURRENT_USER'ADA_LIBRARY(SAMPLE)"
LIST    => "'CURRENT_USER'DOT(LIS)"
LOG     => "'CURRENT_USER'DOT(LOG)"
```

The parameters SOURCE and LIBRARY do not have a default value and need to be specified anyway.

The default of the parameters LIST and LOG means that no listing, resp. no log output is to be produced. The values used for validation are CAIS pathnames in order to obtain the corresponding output in the file nodes specified by the respective pathnames.

Linking was made using the following parameter settings:

```
UNIT       => ...  -- Main Program to be linked
LIBRARY    => "'CURRENT_USER'ADA_LIBRARY(SAMPLE)"
EXECUTABLE => "'CURRENT_USER'DOT(EXE)"
KERNEL     => "'EXECUTABLE_IMAGE'DOT_PARENT'DOT(MTK_133)"
DEBUG      =>  NO
LOG        => "'CURRENT_USER'DOT(LOG)"
```

The parameters UNIT, LIBRARY and EXECUTABLE do not have a default value and need to be specified anyway.

The default value for the parameter KERNEL means that no kernel is linked. The value used for validation is a CAIS pathname to the minimal target kernel for the MVME133XT board. This is the only kernel provided by the customer with the candidate Ada implementation. It does neither contain debugger support nor communication support.

The default of the parameter LOG means that no log output is to be produced. The value used for validation is a CAIS pathname in order to obtain the corresponding output in the file node specified by the pathname.

The default value for the parameter DEBUG is not used, since ALSYS has provided only the runtime system which does not include debugger support.

Test output, compiler and linker listings, and job logs were captured on a Magnetic Tape and archived at the AVF. The listings examined by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	255 -- Value of V
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_647
$DEFAULT_MEM_SIZE	2147483648
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	MOTOROLA_68020_BARE
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	SYSTEM.INTERRUPT_VECTOR(1)
$ENTRY_ADDRESS1	SYSTEM.INTERRUPT_VECTOR(2)
$ENTRY_ADDRESS2	SYSTEM.INTERRUPT_VECTOR(3)
$FIELD_LAST	512
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	0.0
$GREATER_THAN_DURATION_BASE_LAST	200_000.0
$GREATER_THAN_FLOAT_BASE_LAST	16#1.0#E+256
$GREATER_THAN_FLOAT_SAFE_LARGE	16#0.8#E+256
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	16#0.8#E+32

```
$HIGH_PRIORITY          15

$ILLEGAL_EXTERNAL_FILE_NAME1
                        FILE1

$ILLEGAL_EXTERNAL_FILE_NAME2
                        FILE2

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.TST")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006F1.TST")

$INTEGER_FIRST          -2147483648

$INTEGER_LAST           2147483648

$INTEGER_LAST_PLUS_1    2147483648

$INTERFACE_LANGUAGE     ASSEMBLER

$LESS_THAN_DURATION     -0.0

$LESS_THAN_DURATION_BASE_FIRST
                        -200_000.0

$LINE_TERMINATOR        ' '

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        NULL;

$MACHINE_CODE_TYPE      NO_SUCH_TYPE

$MANTISSA_DOC           31

$MAX_DIGITS             18

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2_147_483_648

$MIN_INT                -2147483648

$NAME                   NO_SUCH_TYPE
```

$NAME_LIST	MOTOROLA_68020_BARE
$NAME_SPECIFICATION1	X2120A
$NAME_SPECIFICATION2	X2120B
$NAME_SPECIFICATION3	X3119A
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	2147483648
$NEW_SYS_NAME	MOTOROLA_68020_BARE
$PAGE_TERMINATOR	' '
$RECORD_DEFINITION	NEW INTEGER
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	10240
$TICK	1.0
$VARIABLE_ADDRESS	GET_VARIABLE_ADDRESS
$VARIABLE_ADDRESS1	GET_VARIABLE_ADDRESS1
$VARIABLE_ADDRESS2	GET_VARIABLE_ADDRESS2

APPENDIX B

COMPILATION AND LINKER SYSTEM OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

4 Compiling

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units must all be on the same file. One unit, a parameterless procedure, acts as the main program. If all units needed by the main program and the main program itself have been compiled successfully, they can be linked. The resulting code can then be downloaded by using appropriate tools of the SWG APSE (Loader, Debugger).

§4.1 and Chapter 5 describe in detail how to call the Compiler and the Linker. In §4.2 the Completer, which is called to generate code for instances of generic units, is described.
Chapter 6 explains the information which is given if the execution of a program is abandoned due to an unhandled exception.
The information the Compiler produces and outputs in the Compiler listing is explained in §4.4.
Finally, the log of a sample session is given in Chapter 7.

4.1 Compiling Ada Units

To start the SYSTEAM Ada Compiler, use the `compile_target` command.

compile_target — **Command Description**

Format

```
PROCEDURE compile_target (

    source              : string          :
    analyze_dependency  : yes_no_answer   := no;
    check               : yes_no_answer   := yes;
    copy_source         : yes_no_answer   := yes;
    given_by            : source_choices  := pathname;
    inline              : yes_no_answer   := yes;
    library             : pathname_type
                                  := default_library;
    list                : pathname_type   := nolist;
    log                 : pathname_type   := nolog;
    machine_code        : yes_no_answer   := no;
```

Debugger know this name. You can use the `directory_target (..., full => yes, ...)` command to see the file name of the copy. If a specified file contains several compilation units a copy containing only the source text of one compilation unit is stored in the library for each compilation unit. Thus the Recompiler can recompile a single unit.
If `copy_source => no` is specified, the Compiler only stores the name of the source file in the program library. In this case the Recompiler and the Debugger are able to use the original file if it still exists.
`copy_source => yes` cannot be specified together with `analyze_dependency`.

`given_by : source_choices := pathname`
`given_by => pathname` indicates that the string of the source parameter is to be interpreted as a pathname.
`given_by => unique_identifier` indicates that the string of the source parameter is to be interpreted as a unique identifier.
By default it is interpreted as a pathname.

`inline : yes_no_answer := yes`
Controls whether inline expansion is performed as requested by PRAGMA inline. If you specify no these pragmas are ignored.
By default, inline expansion is performed.

`library : pathname_type := default_library`
Specifies the program library the command works on. The `compile_target` command needs write access to the library.
The default is `'CURRENT_USER'ADA_LIBRARY(STD)`.

`list : pathname_type := nolist`
Controls whether a listing is written to the given file.
By default, the compile command does not produce a listing file.

`log : pathname_type := nolog`
Controls whether the Compiler appends additional messages to the specified file.
By default, no additional messages are written.

`machine_code : yes_no_answer := no`
Controls whether machine code is appended at the listing file. `machine_code` has no effect if `list is nolist` or `analyze_dependency => yes` is specified.
By default, no machine code is appended at the listing file.

`optimize : yes_no_answer := yes`
Controls whether full optimization is applied in generating code. There is no way to specify that only certain optimizations are to be performed.
By default, full optimization is done.

4.2 Completing Generic Instances

Since the Compiler does not generate code for instances of generic bodies, the Completer must be used to complete such units before a program using the instances can be executed. The Completer must also be used to complete packages in the program which do not require a body. This is done implicitly when the Linker is called.

It is also possible to call the Completer explicitly with the `complete_target` command.

complete_target — Command Description

Format

```
PROCEDURE complete_target (

    unit                : unitname_type      ;
    check               : yes_no_answer      := yes;
    inline              : yes_no_answer      := yes;
    library             : pathname_type
                                  := default_library;
    list                : pathname_type      := nolist;
    log                 : pathname_type      := nolog;
    machine_code        : yes_no_answer      := no;
    optimize            : yes_no_answer      := yes);
```

Description

The `complete_target` command invokes the SYSTEAM Ada Completer. The Completer generates code for all instantiations of generic units in the execution closure of the specified unit(s). It also generates code for packages without bodies (if necessary).

By default, the Completer is invoked implicitly by the `link_target` command. In normal cases there is no need to invoke it explicitly.

Parameters

`unit : unitname_type`
specifies the unit whose execution closure is to be completed.

`check : yes_no_answer := yes`
Controls whether all run-time chetks are suppressed. If you specify no this is equivalent to the use of PRAGMA suppress for all kinds of checks.

In the following the term *re*compilation stands for the recompilation of an obsolete unit using the identical source which was used the last time. (This kind of recompilation could alternatively be implemented by using some appropriate intermediate representation of the obsolete unit.) This definition is stronger than that of the LRM (10.3). If a new version of the source of a unit is compiled we call it *compilation*, not a *re*compilation.

The set of units to be checked for recompilation or new compilation is described by specifying a unit and the kind of a closure which is to be built on it. In many cases you will simply specify your main program.

The automatic recompilation of obsolete units is supported by the `recompile_target` command. It determines the set of obsolete units and generates a command file for calling the Compiler in an appropriate order. This command file is in fact an Ada program using the facilities of the package `CLI_INTERFACE` provided by the SWG APSE CLI.

The recompilation is performed using the copy of the obsolete units which is (by default) stored in the library. (If the user does not want to hold a copy of the sources the `recompile_target` command offers the facility to use the original source.)

The automatic compilation of modified sources is supported by the `autocompile_target` command. It determines the set of modified sources and generates a command file for calling the Compiler in an appropriate order. This command file is in fact an Ada program using the facilities of the package `CLI_INTERFACE` provided by the SWG APSE CLI. The basis of both the `recompile_target` and the `autocompile_target` command is the information in the library about the dependencies of the concerned units. Thus neither of these commands can handle the compilation of units which have not yet been entered in the library.

The automatic compilation of new sources is supported by the `compile_target` command together with the `analyze_dependency` parameter. This command is able to accept a set of sources in any order. It makes a syntactical analysis of the sources and determines the dependencies. The units "compiled" with this command are entered into the library, but only their names, their dependencies on other units and the name of the source files are stored in the library. Units which are entered this way can be automatically compiled using the `autocompile_target` command. They *cannot* be recompiled using the `recompile_target` command because the `recompile_target` command only recompiles units which were already compiled.

The next sections explain the usage of the `recompile_target` command, the `autocompile_target` command, and the `compile_target` command with `analyze_dependency => yes`.

The recompile_target command uses the copy of the source which is stored in the library for the recompilation. By default, the compile command stores a copy of the source in the library. If there is no copy in the library - because the unit was compiled using the copy_source => no parameter - the recompile_target issues a warning and generates a compile_target command for the original source file name. It is *not* checked whether such a file still exists. This command only performs a real recompilation if the current source is the same which was last compiled.

In the command file each recompilation of a unit is executed under the condition that the recompilation of other units it depends on was successful. Thus useless recompilations are avoided. The generated command file only works correctly if the library was not modified since the command file was generated.

Note: If a unit from a parent library is obsolete it is compiled in the sublibrary in which the recompile_target command is used. In this case a later recompilation in the parent library may be hidden afterwards.

Parameters

unit : unitname_type
Specifies the unit whose closure is to be built.

output : pathname_type
Specifies the name of the generated command file.

body_ind : yes_no_answer := no
specifies that unit stands for the secondary unit with that name. By default, unit denotes the library unit. If unit specifies a subunit, the body_ind parameter need not be specified.

bodies_only : yes_no_answer := no
Controls whether all units of the closure are recompiled (default) or only the secondary units. This parameter is only effective if conditional => no is specified.

check : yes_no_same_answer := same
check => same means that the same value for the parameter check is included in the generated command file which was in effect at the last compilation. See the same parameter of the compile_target command. Otherwise the given value for the check parameter is included in the command file.
By default the parameter value of the last compilation is included.

closure : closure_choices := execute

Otherwise the given value for the optimize parameter is included in the command file.
By default the parameter value of the last compilation is included.

End of Command Description

4.3.2 Compiling New Sources

The autocompile_target command supports the automatic compilation of units for which a new source exists. The command receives as parameters a set of units which are to be used to form the closure of units to be processed. The kind of closure can be specified. For every unit in the closure, the autocompile_target checks whether there exists a newer source than that which was used for the last compilation. It generates a command file with a sequence of compile_target commands to compile the units for which a newer source exists. If a unit to be compiled depends on another unit which is obsolete or which will become obsolete and for which no newer source exists, the autocompile_target command always adds an appropriate compile_target (..., recompile => yes, ...) command to make it current; the recompile parameter controls which other obsolete units are recompiled, and can indeed be used to specify that the same recompilations are done as if the recompile_target command was applied subsequently. The generated command file is in fact an Ada program using the facilities of the package CLI_INTERFACE provided by the SWG APSE CLI. The name of the command file can be specified using the output parameter.

autocompile_target — Command Description

Format

```
PROCEDURE autocompile_target (

    unit          : unitname_type       ;
    output        : pathname_type       ;
    body_ind      : yes_no_answer       := no;
    bodies_only   : yes_no_answer       := no;
    check         : yes_no_same_answer  := same;
    closure       : closure_choices     := execute;
    conditional   : yes_no_answer       := yes;
    copy_source   : yes_no_answer       := yes;
    inline        : yes_no_same_answer  := same;
    library       : pathname_type
                          := default_library;
```

The autocompile_target command does not fully handle the problem which arises when several compilation units are contained within one source file; it only avoids the multiple compilation of the same source file. If you want to use the autocompile_target command it is recommended not to keep several compilation units in one source.

Parameters

unit : unitname_type
Specifies the unit whose closure is to be built.

output : pathname_type
Specifies the name of the generated command file.

body_ind : yes_no_answer := no
specifies that unit stands for the secondary unit with that name. By default, unit denotes the library unit. If unit specifies a subunit, the body_ind parameter need not be specified.

bodies_only : yes_no_answer := no
Controls whether all new units of the closure are compiled (default) or only the secondary units. This parameter is only effective if conditional => no is specified.

check : yes_no_same_answer := same
check => same means that the same value for the parameter check is included in the generated command file which was in effect at the last compilation. See the same parameter of the compile_target command. Otherwise the given value for the check parameter is included in the command file.
By default the parameter value of the last compilation is included.

closure : closure_choices := execute
Controls the kind of the closure which is built and which is the basis for the investigation for new sources. closure => noclosure means that only the specified unit is to be checked. closure => compile means that only those units on which the specified unit transitively depend(s) are regarded. closure => execute means that - in addition - all related secondary units and the units they depend on are regarded. If closure => tree is specified, a warning is issued stating that this is not meaningful for this command and that the default value is taken instead.
By default, the execution closure is investigated for new sources.

conditional : yes_no_answer := yes
Controls whether the check for new sources is performed (default). no means that all units in the closure are compiled disregarding the modification date. This parameter is useful for compiling the complete closure with different parameters than the last time.

Controls whether the `autocompile_target` command additionally recompiles obsolete units. With `recompile => as_necessary` only those units are recompiled which are obsolete or become obsolete *and* are used by other units which are to be compiled because of new sources. `recompile => same_status` additionally recompiles those units of the considered closure which will become obsolete during the compilation of new sources. This option specifies that there shall not be more obsolete units after the execution of the command file than before. `recompile => as_possible` specifies that all obsolete units of the closure and all units which will become obsolete are recompiled. This is equivalent to a subsequent call of the `recompile_target` command after the run of the command file generated by the `autocompile_target` command.

End of Command Description

4.3.3 First compilation

The SYSTEAM Ada System supports the first compilation of sources for which no compilation order is known by the `compile_target` command with parameter `analyze_dependency` in combination with the `autocompile_target` command.

With the `analyze_dependency` parameter the Compiler accepts sources in any order and performs the syntax analysis. If the sources are syntactically correct the units which are defined by the sources are entered into the library. Their names, their dependencies on other units and the name of the source files are stored in the library. Units which are entered this way can be automatically compiled using the `autocompile_target` command, i.e. the Autocompiler computes the first compilation order for the new sources. The name of the main program, of course, must be known and specified with the `autocompile_target` command.

Note that the `compile_target (..., analyze_dependency => yes, ...)` command replaces other units in the library with the same name as a new one. Thus the library may be modified even if the new units contain semantic errors; but the errors will not be detected until the command file generated by the `autocompile_target` command is run. Hence it is recommended to use an empty sublibrary if you do not know anything about the set of new sources.

If there are several sources containing units with the same name the last analyzed one will be kept in the library.

The `autocompile_target` command issues special warnings if the information about the new units is incomplete or inconsistent.

Warnings and information messages have no influence on the success of a compilation. If there are any other diagnostic messages, the compilation was unsuccessful.

All error messages are self-explanatory. If a source line contains errors, the error messages for that source line are printed immediately below it. The exact position in the source to which an error message refers is marked by a number. This number is also used to relate different error messages given for one line to their respective source positions.

In order to enable semantic analysis to be carried out even if a program is syntactically incorrect, the Compiler corrects syntax errors automatically by inserting or deleting symbols. The source positions of insertions/deletions are marked with a vertical bar and a number. The number has the same meaning as above. If a larger region of the source text is affected by a syntax correction, this region is located for the user by repeating the number and the vertical bar at the end as well, with dots in between these bracketing markings.

A complete Compiler listing follows which shows the most common kinds of error messages, the technique for marking affected regions and the numbering scheme for relating error messages to source positions. It is slightly modified so that it fits into the page width of this document:

```
**************************************************************************
**                                                                      **
** SYSTEAM ADA - COMPILER        VAX/VMS/CAIS x MC68020/BARE  1.82    **
**                                                                      **
** 90-01-29/08:39:44                                                    **
**                                                                      **
**************************************************************************

==========================================================================
=                                                                        =
=                                    Started at  : 08:39:44              =
=                                                                        =
=                                                                        =
=  PROCEDURE  LISTING_EXAMPLE                                            =
=                                                                        =
     1        PROCEDURE listing_example IS
     2        abc : procedure integer RANGE 0 .. 9 := 10E-1;
                    |1.....1|
                                                          1
>>>>>  SYNTAX ERROR
         Symbol(s)  deleted (1)
>>>>>  SYMBOL ERROR (1)    An exponent for an integer literal must not
                           have a minus sign
     3        def integer RANGE 0 .. 9;
```

5 Linking

The Linker of the SYSTEAM Ada System either performs incremental linking or final linking.

Final linking produces a *program image file* (see §5.6) which contains a loadable program. The *code portion* which is part of the program image file must be loaded onto the target later on. Final linking can (but need not) be based on the results of previous incremental linking.

Incremental linking means that a program is linked step by step (say in $N \geq 1$ steps 1 ... N). All steps except the last one are called incremental linking steps. In an incremental linking step, a *collection image file* (see §5.6) containing a *collection* is produced; a collection is a set of Ada units and external units.

Each step $X \in \{2 \ldots N\}$ is based on the result of step X-1. The last step is always a final link, i.e. it links the Ada main program. The result of an incremental linking step is also a code portion which must be loaded onto the target later on.

So the code of a program may consist of several code portions which are loaded onto the target one by one. This is called *incremental loading.*

The reasons for the introduction of the concept of incremental linking and loading into the Ada Cross System are the following:

- It should be possible that some Ada library units and external units are compiled, linked, and burnt into a ROM that is plugged into the target, and that programs using these units are linked afterwards.
- The loading time during program development should be as short as possible. This is achieved by linking those parts of the program that are not expected to be changed (e.g. some library units and the Ada Runtime System). The resulting code portion is loaded to the target and need not be linked or loaded later on. Instead, only those parts of the program that have been modified or introduced since the first link must be linked, so that the resulting coe portion is much smaller in size than the code of the whole program would be. Because typically this code portion is loaded several times during program development, the development cycle time is reduced drastically.

The Runtime System (which is always necessary for the execution of Ada programs) is always linked during the first linking step. In particular, this means that also the version of the Runtime System (Debug or Non-Debug) is fixed during the first step.

The Linker gives the user great flexibility by allowing him to prescribe the mapping of single Ada units and assembler routines into the memory of the target. This, for

link_main_target Command Description

Format

```
PROCEDURE link_main_target (

    unit              : unitname_type      ;
    executable        : pathname_type      ;
    base              : pathname_type      := nobase;
    check             : yes_no_answer      := yes;
    complete          : yes_no_answer      := yes;
    debug             : yes_no_answer      := yes;
    directive         : pathname_type
                                           := default_directive;
    external          : extern_list        := noexternal;
    inline            : yes_no_answer      := yes;
    kernel            : pathname_type      := nokernel;
    library           : pathname_type
                                           := default_library;
    linker_listing    : pathname_type      := nolist;
    list              : pathname_type      := nolist;
    log               : pathname_type      := nolog;
    machine_code      : yes_no_answer      := no;
    map               : pathname_type      := nomap;
    optimize          : yes_no_answer      := yes);
```

Description

The link_main_target command invokes the SYSTEAM Ada Linker for final linking.

The Linker generates a program image and writes it into the file given by the parameter executable. The code portion of this file can be loaded and executed by means of the SWG APSE Loader, resp. SWG APSE Debugger.

Parameters

unit : unitname_type
Specifies the library unit which is the main program. This must be a parameterless library procedure.

executable : pathname_type
Specifies the name of the file which will contain the result of the final link, see §5.6. The named node must not yet exist. It is created by this command.

kernel : pathname_type := nokernel
Specifies the name of the file that contains the assembled code of the Target Kernel that is to be linked to the program.
If kernel=>nokernel is specified, then no Target Kernel is linked to the program. Note that this feature is not meaningful in the SWG APSE.
If you want to link the Minimal Target Kernel for the MVME133XT board to your program, specify kernel=>minimal_kernel. If you want to link one of the target kernels which support the SWG APSE Loader (XTBS Target Kernel) or the SWG APSE Debugger (XSDB Target Kernel) see the corresponding User Manuals of these components.
Note, the Minimal Target Kernel must not be linked to the final program if the Debug Runtime System is used.

library : pathname_type := default_library
Specifies the program library the command works on. The link_main_target command needs write access to the library unless complete=>no is specified. If complete=>no is specified the link_main_target command needs only read access.
The default library is 'CURRENT_USER'ADA_LIBRARY(STD).

linker_listing : pathname_type := nolist
Unless linker_listing => nolist is specified, the Linker of the SYSTEAM Ada System produces a listing file containing a table of symbols which are used for linking the Ada units.
By default, the Linker does not produce a listing file.

list : pathname_type := nolist
This parameter is passed to the implicitly invoked Completer. See the same parameter with the complete_target command.

log : pathname_type := nolog
This parameter controls whether the command writes additional messages onto the specified file, and is also passed to the implicitly invoked Completer. See the same parameter with the complete_target command.

machine_code : yes_no_answer := no
This parameter is passed to the implicitly invoked Completer. See the same parameter with the complete_target command. If linker_listing is not equal to nolist and machine_code => yes is specified, the Linker of the SYSTEAM Ada System generates a listing with the machine code of the program starter in the file given by linker_listing. The program starter is a routine which contains the calls of the necessary elaboration routines and a call for the Ada subprogram which is the main program.
By default, no machine code listing is generated.

map : pathname_type := nomap

This program image file serves as input for the Loader or the Debugger in order to load the code portion included in the file onto the target.

5.2 Linking Collections

A set of Ada units and external units which can be linked separately is called a collection. Such a collection consists on one hand of all compilation units needed by any of the given library units, and on the other hand of all given external units. All compilation units must successfully have been compiled or completed previously.

The code of a linked collection does not contain any unresolved references and can thus be loaded to the target and used by programs linked afterwards without any changes. In particular, this allows the code of a linked collection to be burnt into a ROM. Linking a collection is called incremental linking.

Contrary to final linking, incremental linking is not done selectively. Instead all code and data belonging to the collection is linked, because the Linker does not know which programs or collections will be linked on the collection as a base.

Incremental linking results in a collection image file. There is a code portion in this image file which, together with the code of the given base (if any), is the code of all Ada units and all external (assembler written) units that belong to the collection.

For incremental linking, the Linker is started by the `link_incr_target` command.

link_incr_target — Command Description

Format

```
PROCEDURE link_incr_target (

    collection      : pathname_type      ;
    base            : pathname_type      := nobase;
    contains        : unit_list          := nounits;
    debug           : yes_no_answer      := yes;
    directive       : pathname_type
                                  := default_directive;
    external        : extern_list        := noexternal;
    library         : pathname_type
                                  := default_library;
    log             : pathname_type      := nolog;
    map             : pathname_type      := nomap);
```

Specifies a list of file nodes which contain the object code of those program units which are written in Assembler; these file nodes contain information generated by the Cross Assembler, see §5.5. When `external=>noexternal` is given, external program units are not linked.

`library : pathname_type := default_library`
Specifies the program library the command works on. The `link_incr_target` command needs write access to the library unless `complete=>no` is specified. If `complete=>no` is specified the `link_incr_target` command needs only read access.
The default library is `'CURRENT_USER'ADA_LIBRARY(STD)`.

`log : pathname_type := nolog`
This parameter controls whether the command writes additional messages onto the specified file, and is also passed to the implicitly invoked Completer. See the same parameter with the `complete_target` command.
By default, no additional messages are written.

`map : pathname_type := nomap`
Specifies whether the map listing of the Linker and the table of symbols which are used for linking the Ada units are to be produced in the specified file.

End of Command Description

The `contains` and `external` parameters define a collection C as defined at the beginning of this section. If a base collection is specified (parameter `base`), then C is enlarged by all units belonging to this base collection. The units belonging to the base collection are identified by their names (Ada name of a library unit or name of the external unit) and by their compilation or assembly times. The Linker uses these to check whether a base unit is obsolete or not.

See §5.4 for the mapping process.

If no errors are detected within the linking process, then the result of an incremental link is a collection image file containing the following:

- A code portion that contains the complete code of the linked collection, except the code of the base collection.
- Base addresses and lengths of the regions actually occupied by the complete collection (including the base collection).
- Checksums of the regions which contain code sections and which are actually occupied by the complete collection (including the base collection).
- The names of all library units as specified by the user (parameter `units`) (including those of the base collection).

region_list ::= region_name [(, region_name)+]

The syntax is specified in an extended Backus-Naur notation with start symbol linker_directive_file. [X] means that X is optional, X + means that X is repeated several times (but at least once), X | Y means that X or Y is used.

All characters are case insensitive. Hexadecimal numbers must be in the range 0 .. FFFFFFFF. region_name, library_unit_name, and object_module_name can be any sequence of readable characters except comma and blank.

The user has to specify all contiguous memory regions of the target that are to be used for the program or the collection to be linked. Each REGION description defines the name, the base address, and the size in bytes of one region.

The RESET directive specifies a region whose first 8 bytes are to be reserved for the initial program counter and the initial stack pointer. This directive supports the generation of ROMable programs: If a hardware reset occurs, then the processor fetches its reset vector from the start address of the given region. The RESET directive is ignored if KERNEL is not specified.

The STACK directive tells the Linker the size of the main task's stack and the name of the region into which the stack is to be mapped.

It is possible to specify specific regions for the code or the data of Ada library units or of external (assembler written) units in LOCATION directives. If a region for a library unit is specified, this causes this unit and all its secondary units to be mapped into this region. A region for the code or data of a library unit or of an external unit must not be specified more than once.

In the CODE directive, a list of regions must be specified to be used for the code and the constants of those units for which no LOCATION CODE directive is given. The specified regions are filled in the given order.

In the DATA directive, a list of regions must be specified to be used for the data of those units for which no LOCATION DATA directive is given. The specified regions are filled in the given order.

A list of regions must be given to be used for the heap of the program (HEAP directive).

The following objects are allocated on the heap:

- All Ada collections for which no length clause is specified;
- the storage for a task activation (see §10.3);
- all task control blocks (see §10.3).

Enough space for these objects must be allocated; otherwise `storage_error` will be raised when the heap space is exhausted.

resulting image may be reduced drastically when a base collection is given, even if memory space occupied by the "old" section is not reused.

The Linker automatically takes care that the regions specified by the user in the REGION descriptions of the directive file do not overlap with the regions actually occupied by the given base collection. If the Linker detects an overlap, then it issues an information message and uses a region description which does not overlap with a base region any longer.

The mapping of the sections of S into the regions proceeds as follows:

1. Final linking only: If there is a RESET directive, then space for the initial program counter and for the initial stack pointer is reserved at the bottom of the given region.
2. Final linking only: The stack is mapped into the specified region with the given size. If the given region has not enough space for the stack, then an error message is issued.
3. The LOCATION directives are processed in the order in which they appear in the Linker's directive file. Each directive is treated as follows: If the specified library unit or external unit is not part of the resulting program (e.g. as a consequence of selective linking), then the directive is ignored and a warning is issued. Otherwise all memory sections belonging either to the given library unit or one of its secondary units or to the given external unit, and containing code or data (as given in the directive), are mapped into the given region. If the given region has not enough space left for this mapping, then an error message is issued.
4. Then all sections not yet mapped are processed in an arbitrary order. If a section contains code or constants, then the regions specified in the CODE directive are scanned in the given order and the section is mapped into the first region that has enough space left. If the section is a data section, then the same is done with the regions specified in the DATA directive. If no region is found that has enough space left, then an error message is issued.

Now each region is filled without any gaps, beginning at its base address. The sections which are mapped into a region are sorted as follows: First the stack, then code sections, then data sections. If there is any space left in a region, then this space is a contiguous byte block at the top of the region.

5. Final link only: The heap is located in those regions that have space of at least a certain minimum size (100 byte) left and are listed within the HEAP directive.
6. Final link only: If there is a RESET directive, then the values of the initial stack pointer and of the kernel entry point are written into the first 8 bytes of the given region.

Unless the parameter map=>nomap is given, the result of this mapping is written into the specified map file. The map file is generated even if errors were detected during linking. The information written into the map file has the following structure:

5.6 Image Files

The files whose contents can be downloaded to a target board are called target image files. Two different kinds of target images exist:

- program images
- collection images

Collection images are the result of an incremental link step (command `link_incr_target`), see §5.2. Program images are the result of linking a main program using the command `link_main_target`, see §5.1.

In order to distinguish these kinds of images in a CAIS database from executable images for the host computer the SYSTEAM Ada System defines a node kind `TARGET_IMAGE` and a relation `TARGET_IMAGE` which can terminate at nodes of this kind. Hence, this relation must always be used as the last relation in pathnames denoting target image files of this SYSTEAM Ada System. This applies to pathnames given as values to the parameters `BASE`, `COLLECTION` and `EXECUTABLE` in the commands mentioned.

As an exception to the general rule that nodes have to exist before they can be passed in pathnames to the SYSTEAM Ada System, target image nodes are created, see the description of the parameters `COLLECTION` and `EXECUTABLE` in the respective link commands.

Loading of target image files to a target board is supported by the SWG APSE Loader and SWG APSE Debugger. They take target image files produced by this SYSTEAM Ada System as input.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are contained in the following Predefined Language Enviroment (chapter 13 of the compiler user manual).

13 Predefined Language Environment

The predefined language environment comprises the package standard, the language-defined library units and the implementation-defined library units.

13.1 The Package STANDARD

The specification of the package standard is outlined here; it contains all predefined identifiers of the implementation.

```
PACKAGE standard IS

  TYPE boolean IS (false, true);

  -- The predefined relational operators for this type are as follows:

  -- FUNCTION "="   (left, right : boolean) RETURN boolean;
  -- FUNCTION "/="  (left, right : boolean) RETURN boolean;
  -- FUNCTION "<"   (left, right : boolean) RETURN boolean;
  -- FUNCTION "<="  (left, right : boolean) RETURN boolean;
  -- FUNCTION ">"   (left, right : boolean) RETURN boolean;
  -- FUNCTION ">="  (left, right : boolean) RETURN boolean;

  -- The predefined logical operators and the predefined logical
  -- negation operator are as follows:

  -- FUNCTION "AND" (left, right : boolean) RETURN boolean;
  -- FUNCTION "OR"  (left, right : boolean) RETURN boolean;
  -- FUNCTION "XOR" (left, right : boolean) RETURN boolean;

  -- FUNCTION "NOT" (right : boolean) RETURN boolean;

  -- The universal type universal_integer is predefined.

  TYPE integer IS RANGE - 2_147_483_648 .. 2_147_483_647;

  -- The predefined operators for this type are as follows:

  -- FUNCTION "="  (left, right : integer) RETURN boolean;
  -- FUNCTION "/=" (left, right : integer) RETURN boolean;
```

```
-- FUNCTION "+"   (left, right : float) RETURN float;
-- FUNCTION "-"   (left, right : float) RETURN float;
-- FUNCTION "*"   (left, right : float) RETURN float;
-- FUNCTION "/"   (left, right : float) RETURN float;

-- FUNCTION "**"  (left : float; right : integer) RETURN float;

-- An implementation may provide additional predefined floating
-- point types. It is recommended that the names of such additional
-- types end with FLOAT as in SHORT_FLOAT or LONG_FLOAT.
-- The specification of each operator for the type  universal_real,
-- or for any additional predefined floating point type, is obtained
-- by replacing FLOAT by the name of the type in the specification of
-- the corresponding operator of the type FLOAT.

TYPE short_float IS DIGITS 6 RANGE
        - 16#0.FFFF_FF#E32 .. 16#0.FFFF_FF#E32;

TYPE long_float IS DIGITS 18 RANGE
        - 16#0.FFFF_FFFF_FFFF_FFFF#E4096 ..
          16#0.FFFF_FFFF_FFFF_FFFF#E4096;

-- In addition, the following operators are predefined for universal
-- types:

-- FUNCTION "*" (left : UNIVERSAL_INTEGER; right : UNIVERSAL_REAL)
                RETURN UNIVERSAL_REAL;
-- FUNCTION "*" (left : UNIVERSAL_REAL;    right : UNIVERSAL_INTEGER)
                RETURN UNIVERSAL_REAL;
-- FUNCTION "/" (left : UNIVERSAL_REAL;    right : UNIVERSAL_INTEGER)
                RETURN UNIVERSAL_REAL;

-- The type universal_fixed is predefined.
-- The only operators declared for this type are

-- FUNCTION "*" (left : ANY_FIXED_POINT_TYPE;
                right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;
-- FUNCTION "/" (left : ANY_FIXED_POINT_TYPE;
                right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;

-- The following characters form the standard ASCII character set.
-- Character  literals corresponding to control characters are not
-- identifiers.

TYPE character IS
     (nul,  soh,  stx,  etx,     eot,  enq,  ack,  bel,
```

```
   dollar     : CONSTANT character := '$';
   percent    : CONSTANT character := '%';
   ampersand  : CONSTANT character := '&';
   colon      : CONSTANT character := ':';
   semicolon  : CONSTANT character := ';';
   query      : CONSTANT character := '?';
   at_sign    : CONSTANT character := '@';
   l_bracket  : CONSTANT character := '[';
   back_slash : CONSTANT character := '\';
   r_bracket  : CONSTANT character := ']';
   circumflex : CONSTANT character := '^';
   underline  : CONSTANT character := '_';
   grave      : CONSTANT character := '`';
   l_brace    : CONSTANT character := '{';
   bar        : CONSTANT character := '|';
   r_brace    : CONSTANT character := '}';
   tilde      : CONSTANT character := '~';

   lc_a : CONSTANT character := 'a';
   ...
   lc_z : CONSTANT character := 'z';

END ascii;

-- Predefined subtypes:

SUBTYPE natural  IS integer RANGE 0 .. integer'last;
SUBTYPE positive IS integer RANGE 1 .. integer'last;

-- Predefined string type:

TYPE string IS ARRAY(positive RANGE <>) OF character;

PRAGMA pack(string);

-- The predefined operators for this type are as follows:

-- FUNCTION "="  (left, right : string) RETURN boolean;
-- FUNCTION "/=" (left, right : string) RETURN boolean;
-- FUNCTION "<"  (left, right : string) RETURN boolean;
-- FUNCTION "<=" (left, right : string) RETURN boolean;
-- FUNCTION ">"  (left, right : string) RETURN boolean;
-- FUNCTION ">=" (left, right : string) RETURN boolean;

-- FUNCTION "&" (left : string;    right : string)    RETURN string;
-- FUNCTION "&" (left : character; right : string)    RETURN string;
-- FUNCTION "&" (left : string;    right : character) RETURN string;
```

13.3.1 The Package COLLECTION_MANAGER

In addition to unchecked storage deallocation (cf. LRM(§13.10.1)), this implementation provides the generic package collection_manager, which has advantages over unchecked deallocation in some applications; e.g. it makes it possible to clear a collection with a single reset operation. See §15.10 for further information on the use of the collection manager and unchecked deallocation.

The package specification is:

```
GENERIC
   TYPE elem IS LIMITED PRIVATE;
   TYPE acc  IS ACCESS elem;
PACKAGE collection_manager IS

   TYPE status IS LIMITED PRIVATE;

   PROCEDURE mark (s : OUT status);

      -- Marks the heap of type ACC and
      -- delivers the actual status of this heap.

   PROCEDURE release (s : IN status);

      -- Restore the status s on the collection of ACC.
      -- RELEASE without previous MARK raises CONSTRAINT_ERROR

   PROCEDURE reset;

      -- Deallocate all objects on the heap of ACC

   PRIVATE
      -- private declarations

END collection_manager;
```

A call of the procedure release with an actual parameter s causes the storage occupied by those objects of type acc which were allocated after the call of mark that delivered s as result, to be reclaimed. A call of reset causes the storage occupied by all objects of type acc which have been allocated so far to be reclaimed and cancels the effect of all previous calls of mark.

```
WITH system;

PACKAGE privileged_operations IS

   SUBTYPE long_range IS integer RANGE -2**31 .. 2**31-1;
   SUBTYPE word_range IS integer RANGE -2**15 .. 2**15-1;
   SUBTYPE byte_range IS integer RANGE -2**7  .. 2**7-1;

   SUBTYPE bit_number IS integer RANGE 0 .. 7;
      -- 0 designates the least significant bit of a byte.
      -- 7 designates the most significant bit of a byte.

   SUBTYPE vector_number IS integer RANGE 0 .. 255;


   PROCEDURE assign_byte (dest : system.address;
                          item : byte_range);
   PROCEDURE assign_word (dest : system.address;
                          item : word_range);
   PROCEDURE assign_long (dest : system.address;
                          item : long_range);
   PROCEDURE assign_addr (dest : system.address;
                          item : system.address);

   PROCEDURE bit_set   (dest  : system.address;
                        bitno : bit_number);
   PROCEDURE bit_clear (dest  : system.address;
                        bitno : bit_number);

   FUNCTION byte_value (addr : system.address) RETURN byte_range;
   FUNCTION word_value (addr : system.address) RETURN word_range;
   FUNCTION long_value (addr : system.address) RETURN long_range;
   FUNCTION addr_value (addr : system.address) RETURN system.address;

   FUNCTION bit_value (addr  : system.address;
                       bitno : bit_number) RETURN boolean;
      -- true is returned if the bit is set, false otherwise.

   PROCEDURE define_interrupt_service_routine
                                  (routine    : system.address;
                                   for_vector : vector_number);
      -- defines an assembler routine as interrupt service routine

END privileged_operations;
```

```
   PROCEDURE sincos (x   : long_float;
                     cos : OUT long_float;
                     sin : OUT long_float);

   FUNCTION  sinh   (x : long_float) RETURN long_float;

   FUNCTION  sqrt   (x : long_float) RETURN long_float;

   FUNCTION  tan    (x : long_float) RETURN long_float;

   FUNCTION  tanh   (x : long_float) RETURN long_float;

   FUNCTION  tentox (x : long_float) RETURN long_float;

   FUNCTION  twotox (x : long_float) RETURN long_float;

END coprocessor_interface;
```

15 Appendix F

This chapter, together with the Chapters 16 and 17, is the Appendix F required in the LRM, in which all implementation-dependent characteristics of an Ada implementation are described.

15.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

15.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here.

CONTROLLED
: has no effect.

ELABORATE
: is fully implemented. The SYSTEAM Ada System assumes a PRAGMA elaborate, i.e. stores a unit in the library as if a PRAGMA elaborate for a unit u was given, if the compiled unit contains an instantiation of u (or a generic program unit in u) and if it is clear that u *must* have been elaborated before the compiled unit. In this case an appropriate information message is given. By this means it is avoided that an elaboration order is chosen which would lead to a PROGRAM_ERROR when elaborating the instantiation.

INLINE
: Inline expansion of subprograms is supported with the following restrictions: the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

effect on scheduling - leaving the priority of a task or main program undefined by not giving PRAGMA priority for it is the same as if the PRAGMA priority 0 had been given (i.e. the task has the lowest priority).

SHARED
is fully supported.

STORAGE_UNIT
has no effect.

SUPPRESS
has no effect, but see §15.1.2 for the implementation-defined PRAGMA suppress_all.

SYSTEM_NAME
has no effect.

15.1.2 Implementation-Defined Pragmas

BYTE_PACK
see §16.1.

EXTERNAL_NAME (<string>, <ada_name>)
<ada_name> specifies the name of a subprogram or of an object declared in a library package, <string> must be a string literal. It defines the external name of the specified item. The Compiler uses a symbol with this name in the call instruction for the subprogram. The subprogram declaration of <ada_name> must precede this pragma. If several subprograms with the same name satisfy this requirement the pragma refers to that subprogram which is declared last.
Upper and lower cases are distinguished within <string>, i.e. <string> must be given exactly as it is to be used by external routines. This pragma will be used in connection with the pragma interface (assembler) (see §15.1.1).

15.1.3 Pragma Interface (Assembler,...)

This section describes the internal calling conventions of the SYSTEAM Ada System, which are the same ones which are used for subprograms for which a PRAGMA interface (ASSEMBLER,...) is given. Thus the actual meaning of this pragma is simply that the body needs and must not be provided in Ada, but in object form using the external parameter at link time.

The internal calling conventions are explained in four steps:

- Parameter passing mechanism
- Ordering of parameters
- Type mapping
- Saving registers

Parameter passing mechanism:

The parameters of a call to a subprogram are placed by the caller in an area called *parameter block*. This area is aligned on a longword boundary and contains parameter values (for parameter of scalar types), descriptors (for parameter of composite types) and alignment gaps.
For a function subprogram an extra field is assigned at the beginning of the parameter block containing the function result upon return. Thus the return value of a function is treated like an anonymous parameter of mode OUT. No special treatment is required for a function result except for return values of an unconstrained array type (see below).

A subprogram is called using the JSR instruction. The address pointing to the beginning of the parameter block is pushed onto the stack before calling the subprogram.

In general, the ordering of the parameter values within the parameter block does not agree with the order specified in the Ada subprogram specification. When determining the position of a parameter within the parameter block the calling mechanism and the size and alignment requirements of the parameter type are considered. The size and alignment requirements and the passing mechanism are described in the following:

Scalar parameters or parameters of access types are passed by value, i.e. the values of the actual parameters of modes IN or IN OUT are copied into the parameter block before the call. Then, after the subprogram has returned, values of the actual parameters of modes IN OUT and OUT are copied out of the parameter block into the associated actual parameters. The parameters are aligned within the parameter block according to their size: A parameter with a size of 8, 16 or 32 bits (or a multiple of 8 bits greater than 32) has an alignment of 1, 2 or 4 (which means that the object is aligned to a byte, word or longword boundary within the parameter block). If the size of the parameter is not a multiple of 8 bits (which may be achieved by attaching

requirements of a parameter it is not always possible to place parameters in such a way that two consecutive parameters are densely located in the parameter block. In such a situation a gap, i.e. a piece of memory space which is not associated with a parameter, exists between two adjacent parameters. Consequently, the size of the parameter block will be larger than the sum of the sizes used for all parameters. In order to minimize the size of the gaps in a parameter block an attempt is made to fill each gap with a parameter that occurs later in the parameter list. If during the allocation of space within the parameter block a parameter is encountered whose size and alignment fit the characteristics of an available gap, then this gap is allocated for the parameter instead of appending it at the end of the parameter block. As each parameter will be aligned to a byte, word or longword boundary the size of any gap may be one, two or three bytes. Every gap of size three bytes can be treated as two gaps, one of size one byte with an alignment of 1 and one of size two bytes with an alignment of 2. So, if a parameter of size two is to be allocated, a two byte gap, if available, is filled up. A parameter of size one will fill a one byte gap. If none exists but a two byte gap is available, this is used as two one byte gaps. By this first fit algorithm all parameters are processed in the order they occur in the Ada program.

A called subprogram accesses each parameter for reading or writing using the parameter block address incremented by an offset from the start of the parameter block suitable for the parameter. So the value of a parameter of a scalar type or an access type is read (or written) directly from (into) the parameter block. For a parameter of a composite type the actual parameter value is accessed via the descriptor stored in the parameter block which contains a pointer to the actual object. When standard entry code sequences are used within the assembler subprogram (see below), the parameter block address is accessible at address 8(A6).

Type mapping:

To access individual components of array or record types, knowledge about the type mapping for array and record types is required. An array is stored as a sequential concatenation of all its components. Normally, pad bits are used to fill each component to a byte, word, longword or a multiple thereof depending on the size and alignment requirements of the components' subtype. This padding may be influenced using one of the PRAGMAs pack or byte_pack (cf. §16.1). The offset of an individual array component is then obtained by multiplying the padded size of one array component by the number of components stored in the array before it. This number may be determined from the number of elements for each dimension using the fact that the array elements are stored row by row. (For unconstrained arrays the number of elements for each dimension can be found in the descriptor stored in the parameter block.)

A record object is implemented as a concatenation of its components. Initially, locations are reserved for those components that have a component clause applied to them. Then locations for all other components are reserved. Any gaps large enough to hold components without component clauses are filled, so in general the record components are rearranged. Components in record variants are overlaid. The ordering mechanism

for procedures without parameters and

```
RTD     #4
```

for functions and procedures with parameters.

Consider the following example. A function sin is to be implemented by an assembler routine. Its Ada specification is as follows:

```
FUNCTION sin (x : long_float) RETURN long_float;
   PRAGMA interface (assembler, sin);
   PRAGMA external_name ("CPSIN", sin);
```

It is implemented by the following assembler routine:

```
CPSIN:   LINK.W     A6,#-4        *-- allocate frame
         CLR.L      (-4,A6)       *-- clear the indicator bits
         MOVEA.L    (8,A6),A0     *-- address of parameter block
         FSIN.X     (12,A0),FP0   *-- parameter x
         FMOVE.X    FP0,(A0)      *-- store function result
         UNLK       A6            *-- remove frame
         RTD        #4            *-- return to caller
```

15.2 Implementation-Dependent Attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this section.

The value delivered by this attribute applied to a task type or task object is as follows:
If a length specification (STORAGE_SIZE, see §16.2) has been given for the task type, the attribute delivers that specified value; otherwise, the default value is returned.

15.2.2 Implementation-Defined Attributes

There are no implementation-defined attributes.

15.3 Specification of the Package SYSTEM

The package system as required in the LRM(§13.7) is reprinted here with all implementation-dependent characteristics and extensions filled in.

```
PACKAGE system IS

   TYPE designated_by_address IS LIMITED PRIVATE;

   TYPE address IS ACCESS designated_by_address;
   FOR address'storage_size USE 0;

   address_zero : CONSTANT address := NULL;

   FUNCTION "+" (left : address; right : integer) RETURN address;
   FUNCTION "+" (left : integer; right : address) RETURN address;
   FUNCTION "-" (left : address; right : integer) RETURN address;
   FUNCTION "-" (left : address; right : address) RETURN integer;

   SUBTYPE external_address IS STRING;

   -- External addresses use hexadecimal notation with characters
   -- '0'..'9', 'a'..'f' and 'A'..'F'. For instance:
   --    "7FFFFFFF"
   --    "80000000"
   --    "8" represents the same address as "00000008"

   FUNCTION convert_address (addr : external_address) RETURN address;
   -- convert_address raises CONSTRAINT_ERROR if the external address
```

```
mode_error_id        : CONSTANT exception_id := ...;
name_error_id        : CONSTANT exception_id := ...;
use_error_id         : CONSTANT exception_id := ...;
device_error_id      : CONSTANT exception_id := ...;
end_error_id         : CONSTANT exception_id := ...;
data_error_id        : CONSTANT exception_id := ...;
layout_error_id      : CONSTANT exception_id := ...;
time_error_id        : CONSTANT exception_id := ...;

no_error_code      : CONSTANT := 0;

TYPE exception_information
     IS RECORD
        excp_id           : exception_id;
           -- Identification of the exception. The codings of
           -- the predefined exceptions are given above.
        code_addr         : address;
           -- Code address where the exception occurred. Depending
           -- on the kind of the exception it may be be address of
           -- the instruction which caused the exception, or it
           -- may be the address of the instruction which would
           -- have been executed if the exception had not occurred.
        error_code        : integer;
    END RECORD;

PROCEDURE get_exception_information
             (excp_info : OUT exception_information);
-- The subprogram get_exception_information must only be called
-- from within an exception handler BEFORE ANY OTHER EXCEPTION
-- IS RAISED. It then returns the information record about the
-- actually handled exception.
-- Otherwise, its result is undefined.

PROCEDURE raise_exception_id
             (excp_id : exception_id);

PROCEDURE raise_exception_info
             (excp_info : exception_information);

-- The subprogram raise_exception_id raises the exception
-- given as parameter. It corresponds to the RAISE statement.

-- The subprogram raise_exception_info raises the exception
-- described by the information record supplied as parameter.
-- In addition to the subprogram raise_exception_id it allows to
-- explicitly define all components of the exception information
-- record.
```

15.7 Restrictions on Unchecked Conversions

The implementation supports unchecked type conversions for all kinds of source and target types with the restriction that the target type must not be an unconstrained array type. The result value of the unchecked conversion is unpredictable, if

```
target_type'SIZE > source_type'SIZE
```

15.8 Characteristics of the Input-Output Packages

The implementation-dependent characteristics of the input-output packages as defined in the LRM(Chapter 14) are reported in Chapter 17 of this manual.

15.9 Requirements for a Main Program

A main program must be a parameterless library procedure. This procedure may be a generic instantiation; the generic procedure need not be a library unit.

15.10 Unchecked Storage Deallocation

The generic procedure `unchecked_deallocation` is provided; the effect of calling an instance of this procedure is as described in the LRM(§13.10.1).

The implementation also provides an implementation-defined package `collection_manager`, which has advantages over unchecked deallocation in some applications (cf. §13.3.1).

Unchecked deallocation and operations of the `collection_manager` can be combined as follows:

- `collection_manager.reset` can be applied to a collection on which unchecked deallocation has also been used. The effect is that storage of all objects of the collection is reclaimed.
- After the first `unchecked_deallocation` (`release`) on a collection, all following calls of `release` (`unchecked deallocation`) until the next `reset` have no effect, i.e. storage is not reclaimed.

16 Appendix F: Representation Clauses

In this chapter we follow the section numbering of Chapter 13 of the LRM and provide notes for the use of the features described in each section.

16.1 Pragmas

PACK

As stipulated in the LRM(§13.1), this pragma may be given for a record or array type. It causes the Compiler to select a representation for this type such that gaps between the storage areas allocated to consecutive components are minimized. For components whose type is an array or record type the PRAGMA PACK has no effect on the mapping of the component type. For all other component types the Compiler will choose a representation for the component type that needs minimal storage space (packing down to the bit level). Thus the components of a packed data structure will in general not start at storage unit boundaries.

BYTE_PACK

This is an implementation-defined pragma which takes the same argument as the predefined language PRAGMA PACK and is allowed at the same positions. For components whose type is an array or record type the PRAGMA BYTE_PACK has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. But in contrast to PRAGMA PACK all components of a packed data structure will start at storage unit boundaries and the size of the components will be a multiple of `system.storage_unit`. Thus, the PRAGMA BYTE_PACK does not effect packing down to the bit level (for this see PRAGMA PACK).

16.4 Record Representation Clauses

Record representation clauses are supported. The value of the expression given in an alignment clause must be 0, 1, 2 or 4. If this restriction is violated, the Compiler responds with a RESTRICTION error message in the Compiler listing. If the value is 0 the objects of the corresponding record type will not be aligned, if it is 1, 2 or 4 the starting address of an object will be a multiple of the specified alignment.

The number of bits specified by the range of a component clause must not be greater than the amount of storage occupied by this component. (Gaps between components can be forced by leaving some bits unused but not by specifying a bigger range than needed.) Violation of this restriction will produce a RESTRICTION error message.

There are implementation-dependent components of record types generated in the following cases :

- If the record type includes variant parts and if it has either more than one discriminant or else the only discriminant may hold more than 256 different values, the generated component holds the size of the record object.
- If the record type includes array or record components whose sizes depend on discriminants, the generated components hold the offsets of these record components (relative to the corresponding generated component) in the record object.

But there are no implementation-generated names (cf. LRM(§13.4(8))) denoting these components. So the mapping of these components cannot be influenced by a representation clause.

16.5 Address Clauses

Address clauses are supported for objects declared by an object declaration and for single task entries. If an address clause is given for a subprogram, package or a task unit, the Compiler responds with a RESTRICTION error message in the Compiler listing.

If an address clause is given for an object, the storage occupied by the object starts at the given address. Address clauses for single entries are described in §16.5.1.

_IRRETURN is defined by the Ada Runtime System. It contains the start address of the Target Kernel routine that carries out the return from interrupt handling. It is very important that when leaving ISR all registers (except the status register) have the same values as they had when entering ISR. ISR is executed in the supervisor state of the processor, so all instructions (including privileged ones) can be used within ISR. The processor's priority depends on the interrupt source.

If you want to call the interrupt entry with the number N, then you must set a bit within the interrupt entry call pending indicator _IRENTRYC by the instruction:

```
bfset    (_IRRENTRYC).1.{N-1:1}  *-- prepare call of
                                 *-- interrupt entry N
```

(Note: The Microware Cross Assembler has a bug which causes wrong code to be generated for the bfset. The example shows a work around for this bug.)
This instruction should be placed immediately in front of the last instruction of ISR. ISR need not call the interrupt entry each time it is activated. Instead ISR can, for example, read one character each time it is activated, but call the interrupt entry only when a complete line has been read.

A complete example for interrupt handling follows. For this example the second RS232 serial line of the MVME133XT board is used (available through the P2 connector). The assembler routine ISR_READ is activated each time a character is received on that line. ISR_READ calls interrupt entry char_entry of TASK terminal_in. terminal_in uses TASK terminal_out to output each character read.
The terminal should be set up for XON/XOFF (not CTS/RTS) flow control.

```
WITH system,
     privileged_operations,
     text_io;

USE  privileged_operations,
     text_io;

PROCEDURE terminal IS

   PRAGMA priority (2);

   PROCEDURE setup_scc;
      PRAGMA interface (assembler, setup_scc);
      PRAGMA external_name ("SETUP_SCC", setup_scc);

   PROCEDURE isr_read;
      PRAGMA interface (assembler, isr_read);
      PRAGMA external_name ("ISR_READ", isr_read);
```

```
      END LOOP;
   END terminal_out;

BEGIN
   setup_scc;

   define_interrupt_service_routine
                     (isr_read'address, 16#80#);
END terminal;
```

The following assembler routines also belong to the example:

```
typlang  equ 0
attrrev  equ $8000
*--
         psect    terminal,typlang,attrrev,0,0,0
*--
sccb_rro equ      $FFFA0000
sccb_wro equ      $FFFA0000
sccb_rdr equ      $FFFA0001
sccb_tdr equ      $FFFA0001
*--
SETUP_SCC:
   move.b   #$30,(sccb_wro).l    *-- clear receiver error status
   move.b   #$10,(sccb_wro).l    *-- clear external status interrupts
   move.b   #$09,(sccb_wro).l    *-- WR 9
   move.b   #$40,(sccb_wro).l    *-- reset channel A & B, disable IRs
*--
   move.b   #$0A,(sccb_wro).l    *-- WR 10
   move.b   #$00,(sccb_wro).l    *-- NRZ format
   move.b   #$0E,(sccb_wro).l    *-- WR 14
   move.b   #$82,(sccb_wro).l    *-- source=BR generator, RTXC input
*--                              *-- disable BR generator
   move.b   #$04,(sccb_wro).l    *-- WR 4
   move.b   #$44,(sccb_wro).l    *-- clck mode=x16,1 stop bit,no parity
   move.b   #$03,(sccb_wro).l    *-- WR 3
   move.b   #$C1,(sccb_wro).l    *-- 8 bits, enable receiver
   move.b   #$05,(sccb_wro).l    *-- WR 5
   move.b   #$EA,(sccb_wro).l    *-- DTR&RTS=on,8 bits,enable transmtr
   move.b   #$0C,(sccb_wro).l    *-- WR 12
   move.b   #$02,(sccb_wro).l    *-- lower byte of time const (9600 Bd)
   move.b   #$0D,(sccb_wro).l    *-- WR 13
   move.b   #$00,(sccb_wro).l    *-- upper byte of time const (9600 Bd)
   move.b   #$0B,(sccb_wro).l    *-- WR 11
   move.b   #$56,(sccb_wro).l    *-- RxClock=TxClock=TRxClock=BR output
*--                              *-- TRxC output
```

17 Appendix F: Input-Output

In this chapter we follow the section numbering of Chapter 14 of the LRM and provide notes for the use of the features described in each section.

17.1 External Files and File Objects

The implementation only supports the files standard_input and standard_output of PACKAGE text_io. Any attempt to create or open a file raises the exception use_error.

17.2 Sequential and Direct Files

Sequential and direct files are not supported.

17.3 Text Input-Output

standard_input and standard_output are associated with the RS232 serial port of the target.

If the Minimal Target Kernel is used, then this serial port is used and all data of standard_output is directly written to this port and all data of standard_input is directly read from this port.

If the XTBS or XSDB Target Kernel is used, see the corresponding user manual for the behaviour of the text input/output.

For tasking aspects of I/O operations see Chapter 14.

For further details on the I/O implementation within the Target Kernel see Chapter 19.